A Theorem Proving Approach to Analysis of Secure Information Flow

Ádám Darvas, Reiner Hähnle, and David Sands

Chalmers University of Technology & Göteborg University
Department of Computing Science
S-41296 Göteborg, Sweden
{darvas,reiner,dave}@cs.chalmers.se

Abstract. Most attempts at analysing secure information flow in programs are based on domain-specific logics. Though computationally feasible, these approaches suffer from the need for abstraction and the high cost of building dedicated tools for real programming languages. We recast the information flow problem in a general program logic rather than a problem-specific one. We investigate the feasibility of this approach by showing how a general purpose tool for software verification can be used to perform information flow analyses. We are able to handle phenomena like method calls, loops, and object types for the target language Java Card. We are also able to prove insecurity of programs.

1 Introduction

Most attempts at analysing secure information flow in programs have followed basically the same pattern: information flow is modeled using a domain-specific logic (such as a type system or dataflow analysis framework) with a predefined degree of approximation, and this leads to a fully automated but approximate analysis of information flow. There are two problems stemming from this approach. Firstly, the degree of approximation in the logic is fixed and thus secure programs will be rejected unless they can be rewritten. Secondly, implementing a domain-specific tool for a real programming language is a substantial undertaking, and thus there are very few real-language tools available [11].

This paper takes a first step towards an alternative approach based on a general theorem prover:

– We recast the information flow problem in a general program logic rather than a problem-specific one. Program logics based on simple safety and liveness properties (e.g. Hoare logic or weakest precondition calculus) are inadequate for this purpose, since information flow properties cannot be expressed as a simple conjunction of safety and liveness properties[1]. Our approach is to use dynamic logic, which admits a simple characterisation of secure information flow for deterministic programs.

[1] This claim is not formal, since it depends on precisely what one means by “safety” and “liveness”; for some concrete instances see [13, 9, 10].

– We investigate the feasibility of the approach by showing how a general purpose tool for software verification (based on dynamic logic) can be used to perform information flow analyses. So far, our examples are relatively small, but we are able to handle phenomena like method calls, loops, and object types. We are also able to prove insecurity of programs.

2 Modeling Secure Information Flow in Dynamic Logic

2.1 A Dynamic Logic for Java Card

The platform for our experiments is the KeY tool [1], an integrated tool for development *and* verification of object-oriented programs. Among other things, it features an interactive theorem prover for formal verification of JAVA CARD programs. In KeY, the target program to be verified and its specification are both modeled in an instance of dynamic logic (DL) [6] called JAVA CARD DL [3].

JAVA CARD DL generalizes variants of DL used so far for theoretical investigations [6] or verification purposes [2], because it handles such phenomena as side effects, aliasing, object types, exceptions, as partly explained below. Other programming languages than JAVA CARD could be axiomatized in DL. Once this is done, the KeY tool can then be used on them.

Like other interactive theorem provers for software verification, the proving process in KeY is partially automated by heuristic control of applicable rules.

Deduction in JAVA CARD DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA. It can be seen as a modal logic with a modality $\langle \mathtt{p} \rangle$ for every program `p`, where $\langle \mathtt{p} \rangle$ refers to the state (if `p` terminates) that is reached by running program `p`.

The *program formula* $\langle \mathtt{p} \rangle \phi$ expresses that the program `p` terminates in a state in which ϕ holds. A formula $\phi \rightarrow \langle \mathtt{p} \rangle \psi$ is valid if for every state s satisfying pre-condition ϕ a run of the program `p` starting in s terminates, and in the terminating state the post-condition ψ holds.

Thus, the DL formula $\phi \rightarrow \langle \mathtt{p} \rangle \psi$ is similar to the total-correctness Hoare triple $\{\phi\}\,\mathtt{p}\,\{\psi\}$ or to ϕ implying the weakest precondition of `p` wrt ψ. But in contrast to Hoare logic and weakest precondition calculus (wpc), the set of formulas of DL is closed under the usual logical operators and first order quantifiers. For example, in Hoare logic and wpc the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. In general, program formulas can appear anywhere in DL as subformulas.

The programs in JAVA CARD DL formulas are basically executable JAVA CARD code. Each rule of the calculus for JAVA CARD DL specifies how to execute one particular statement, possibly with additional restrictions. When a loop or a recursive method call is encountered, it is necessary to perform induction over a suitable data structure.

In JAVA (like in other object-oriented programming languages), different object variables can refer to the same object. This phenomenon, called aliasing, causes serious difficulties for handling of assignments in a calculus for JAVA CARD DL.

For example, whether or not a formula "`o1.a` = `1`;" still holds after the (symbolic) execution of the assignment "`o2.a` = `2`;", depends on whether or not `o1` and `o2` refer to the same object.

Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution. In JAVA CARD DL calculus a different solution is used, based on the notion of (state) *updates*. These updates are of the form $\{\texttt{loc} := val\}$ and can be put in front of any formula. The semantics of $\{\texttt{loc} := val\}\phi$ is the same as that of $\langle \texttt{loc} = \texttt{val};\rangle\phi$. The difference between an update and an assignment is syntactical. The expressions `loc` and *val* must be simple in the following sense: `loc` is either (i) a program variable `var`, or (ii) a field access `obj.attr`, or (iii) an array access `arr[i]`; and *val* is a logical term (that is free of side effects). More complex expressions are not allowed in updates.

The syntactical simplicity of `loc` and *val* has semantical consequences. In particular, computing the value of *val* has no side effects. The KeY system has simplification rules to compute the result of applying an update to logical terms and program-free formulas. Computing the effect of an update on any program `p` (that is, a formula $\langle \texttt{p}\rangle\phi$) is delayed until `p` was symbolically executed using other rules of the calculus. Thus, case distinctions on object identity are not merely delayed, but can often be avoided altogether, because (i) updates are simplified *before* their effect is computed and (ii) their effect is computed when maximal information is available (after symbolic execution of the program).

There is another important usage of updates. In JAVA CARD DL there are two different types[2] of variables: *program (local) variables* and *logic variables*. Program variables can occur in program parts of a formula as well as outside program parts. Syntactically, they are constants of the logic. Their semantic interpretation depends on the program execution state. Logic variables occur only bound (quantified) and never in programs. Syntactically, they are variables of the logic. Their semantic interpretation is *rigid*, that is, independent of the program state. This is necessary for being able to store previous execution states. Hence, in JAVA CARD DL quantification over program variables like "$\forall \texttt{x}.\, \langle \texttt{p}[\texttt{x}]\rangle\psi[\texttt{x}]$" is syntactically illegal[3].

Updates remedy this problem. Suppose we want to quantify over `x` of type integer. We declare an integer program variable `px`, quantify over a logic variable of type integer lx, and use an update to assign the value of lx to `px`:

$$\langle \texttt{int px;}\rangle\, (\forall lx : int.\, \{\texttt{px} := lx\}\langle \texttt{p}[\texttt{px}]\rangle\psi[lx, \texttt{px}]) \tag{1}$$

2.2 Secure Information Flow Expressed in Dynamic Logic

We use the greater expressivity of DL as compared to Hoare logic and wpc to give a very natural logic modeling of secure information flow. Let `l` be the

[2] Ultimately, this distinction of variables is demanded by side effects in imperative programming languages like JAVA. It is not present in "pure" DL [6].

[3] To stress occurence of variables in formulas or programs, we use the notion `p[x]`, etc.

low-security variables of program p and h the high-security ones. We want to express that by observing the initial and final values of l, it is impossible to know anything about the initial value of h [5]. In other words:

> "When starting p with arbitrary values l, then the value r of l after executing p, is independent of the choice of h."

This can be directly formulated in standard DL:

$$\forall \mathtt{l}.\ \exists r.\ \forall \mathtt{h}.\ \langle \mathtt{p} \rangle\ r \doteq \mathtt{l} \tag{2}$$

To illustrate our formulation in Java Card DL, assume that all variables are of type integer and program variables and logic variables are prefixed by "p" and "l", respectively. Using (1), we obtain the formula:

$$\langle \mathtt{int\ pl;\ int\ ph;} \rangle\, (\forall\, ll{:}int.\, \exists\, r{:}int.\, \forall\, lh{:}int.\, \{\mathtt{pl} := ll\}\{\mathtt{ph} := lh\}\langle \mathtt{p} \rangle\, r \doteq \mathtt{pl}) \tag{3}$$

For sake of readability we use the simpler DL notation (2) in the rest of the paper, unless the actual Java Card DL formulation is of interest.

With the choice of DL, we exploit that one can quantify over variables occurring anywhere in the assertions. Joshi and Leino [7, Cor. 3] arrive at a similar formulation in Hoare logic, but there it is necessary to provide a concrete function for the values of r. Moreover, their characterization assumes that p terminates. In DL we can easily express the additional requirement that no information on the value of h shall be leaked by p's termination behaviour:

$$\forall\, \mathtt{l}.\ (\exists\, \mathtt{h}.\, \langle \mathtt{p} \rangle\, \mathrm{true} \rightarrow \exists\, r.\, \forall\, \mathtt{h}.\, \langle \mathtt{p} \rangle\, r \doteq \mathtt{l}) \tag{4}$$

In addition to (2) this expresses that, for any choice of l, if p terminates for some initial value of h, then it terminates for all values.

3 Interactive Proving of Secure Information Flow

In our experiments, we considered only problems of the form (2) (we could have used form (4), but our examples are all terminating, so the antecedent of the implication is true for all values of h).

In this section, we first show our approach on small programs taken from the literature. Then a program containing a `while` loop will be examined; finally we demonstrate with a somewhat bigger example how the approach works for programs with object types.

3.1 Simple Programs

We demonstrate the feasibility of our approach with some examples taken from papers [7, 11]. Table 1 shows the example programs with the corresponding number of rules applied in the KeY system and the required user interaction, if any. Note that l, h, and r are single variables in each case.

program	*rules applied*	*user interactions*
`l=h;`	7	–
`h=l;`	10	instantiation
`l=6;`	10	instantiation
`l=h; l=6;`	11	instantiation
`h=l; l=h;`	11	instantiation
`l=h; l=l-h;`	12	instantiation
`if (false) l=h;`	10	instantiation
`if (h>=0) l=1; else l=0;`	21	–
`if (h==1) l=1; else l=0; l=0;`	29	instantiation

Table 1. Example programs.

When evaluating the data one must keep in mind that we used the KeY prover as it comes. The KeY system features so-called *taclets*, a simple, yet powerful mechanism by virtue of which users can extend the prover with application specific rules and heuristics.

The user interaction *instantiation* in Table 1 means a single quantifier elimination by supplying a suitable instance term. In the KeY system, the user can simply drag-and-drop the desired term from any place in the current goal over a rule application. In the examples, one mainly has to specify a Skolem term or a constant (e.g., 6 in program "$\mathtt{l} = \mathtt{6}$;") to instantiate the result value r in (2). At the time, when r must be instantiated, this kind of interaction could mostly be eliminated by heuristics that perform instantiation automatically, when there is only one candidate.

Not counting the time for user interactions, all proofs are obtained within fractions of a second.

If a program is secure, then the DL formula (2) is provable. For insecure programs the proof cannot be completed, and there will be one or more *open goal*. Among our examples there are two insecure programs. Table 2 contains the goals (in this case one for each program) that remain open in an attempt to prove security of these programs in KeY. It is easy to observe that these formulas are not provable. In fact, the open goals give a direct hint to the source of the security breach.

It is important to note that the number of applied rules and user interactions does not increase more than linearly if we take the composition of two programs. For example, to verify security of the program "$\mathtt{h} = \mathtt{l}$; $\mathtt{l} = \mathtt{6}$;", one instantiation is required, and the prover applies 11 rules. By comparison, to prove security of the constituents "$\mathtt{h} = \mathtt{l}$;" and "$\mathtt{l} = \mathtt{6}$;", one instantiation and 10 rule applications are used in each case.

3.2 Proving Insecurity

To prove that the programs in Table 2 are insecure, the insecurity property has to be formalized. This can be done by simply taking the negation of formula (2).

program	*open goal*
`l=h;`	$\exists r.\, \forall \mathtt{h}.\; r \doteq \mathtt{h}$
`if (h>=0) l=1; else l=0;`	$\exists r.\, (\forall \mathtt{h}.\, (!(\mathtt{h} < 0) \to r \doteq 1)$ $\&\; \forall \mathtt{h}.\, (\mathtt{h} < 0 \to r \doteq 0))$

Table 2. Open goals for insecure programs.

program	*rules applied*	*user interactions*
`l=h;`	13	instantiation
`if (h>=0) l=1; else l=0;`	34	arithmetic, instantiation

Table 3. Proving insecurity.

The syntactic closure property of DL is crucial again here. Negating (2) and straightforward simplification yields:[4]

$$\exists\, \mathtt{l}.\, \forall\, r.\, \exists \mathtt{h}.\, \langle \mathtt{p} \rangle\, r \neq \mathtt{l} \tag{5}$$

The intuitive meaning of the formula is the following:

> "There is an initial value `l`, such that for any possible final value r of `l` after executing `p`, there exists an initial value `h` which can prevent `l` from taking that final value r."

Table 3 contains the corresponding data of the proofs of insecurity. The user interaction *arithmetic* means that the user has to apply (few) rules manually to close subgoals containing arithmetic properties (e.g. $\forall\, a.\, (a \doteq 0 \to a \neq 1)$). At the moment these cannot be handled automatically by the prover.

3.3 Loops

In this section we report on an experiment with a program containing a `while` loop. The DL formula is the following:

$$\forall\, \mathtt{l}.\, \exists\, r.\, \forall\, \mathtt{h}.\, (\mathtt{h} > 0 \to \langle \mathtt{while}\ (\mathtt{h} > 0)\ \{\mathtt{h}{-}{-};\ \mathtt{l} = \mathtt{h};\}\rangle \mathtt{l} \doteq r) \tag{6}$$

The loop contains the insecure statement $\mathtt{l} = \mathtt{h}$; but the condition of exiting the loop is $\mathtt{h} \doteq 0$, thus the final value of `l` is always 0, independently of the initial value of `h`.

To prove properties of programs containing loops requires in general to perform induction over a suitable *induction variable.* Finding the right induction hypothesis is not an easy task, but once it is found, completing the proof is usually a mechanical process; if one runs into problems, this is a hint, that the

[4] We do actually a little more than is required by proving termination of `p`. Formula (5) really is the negation of (4).

hypothesis was not correct. Heuristic techniques to find induction hypotheses are available in the literature and will be built into KeY in due time.

After the induction hypothesis is given to the prover, three open goals must be proven: (i) after exiting the loop, the post condition holds (induction base), (ii) the induction step, (iii) the induction hypothesis implies the original subgoal.

To prove security of (6), the prover took 163 steps; in addition to establishing the induction hypothesis, several kinds of user interactions were required: instantiation, unwinding the loop, and arithmetic.

The proof cannot be quite completed with the current version of the KeY tool, because the update simplifier is not (yet) powerful enough. In order to show that this is in fact merely a technical problem, we outline the problem in detail. One must be able to prove that the following two program states (expressed by the corresponding updates) are identical:

$$\begin{array}{ll} \{\mathtt{pl} := c_1 + 1\} & \{\mathtt{pl} := c_2\} \\ \quad \{\mathtt{ph} := c_1 + 1\} & \quad \{\mathtt{ph} := c_1 + 1\} \\ & \qquad \{\mathtt{pl} := \mathtt{ph}\} \end{array}$$

The bottommost update is the most recent update (the sequence of updates parallels the sequence of assignments that led to their creation). The c_i are Skolem constants.

3.4 Using Object Types

Next we demonstrate that our approach applies to an object-oriented setting in a natural way. The example presented here is taken from [8, Fig. 5.], where an object (specified by its statechart diagram) leaks information of a high variable through one of its operations. The corresponding Java implementation is:

```
public class Account {
  int balance;
  boolean extraService;

  public void writeBalance(int amount) {
    if (amount>=10000) extraService=true; else extraService=false;
    balance=amount;
  }

  public int readBalance() {return balance;}

  public boolean readExtra() {return extraService;}
}
```

The *balance* of an Account object can be written by the method `writeBalance` and read by `readBalance`. If the balance is over 10000, variable *extraService* is set to true, otherwise to false. The state of that variable can be read by readExtra. The balance of the account and the return value of readBalance are secure, whereas the value of extraService is not.

The program is insecure, since partial information about the high-security variable can be inferred via the observation of a low-security variable. That is, calling writeBalance with different parameters can lead to different observations of the return value of readExtra.

To prove insecurity of this program, we continue to use (2,5). We give the actual JAVA CARD DL formula of security to show how naturally objects are woven into the logic. Where necessary, we use variables with object types in the logic (a detailed account on how to render object types in first-order logic is [4]).

$$\begin{array}{l}\langle \texttt{Account o; int amount; boolean result;}\rangle \\ \quad \forall\, lextraService : boolean.\, \exists\, r : boolean.\, \forall\, lamount : int. \\ \quad\quad \{\texttt{o.extraService} := lextraService\}\{\texttt{amount} := lamount\} \\ \quad\quad\; \langle \texttt{o.writeBalance(amount); result = o.readExtra();}\rangle r \doteq \texttt{result}\end{array}$$

The prover applies 62 rules and stops at the unprovable open goal:

$$\begin{array}{l}\exists\, r : boolean.\, (\forall\, lamount : int.\, (!(lamount < 10000) \to r \doteq \texttt{TRUE})\ \& \\ \qquad\qquad \forall\, lamount : int.\, (lamount < 10000 \to r \doteq \texttt{FALSE}))\end{array}$$

Insecurity of the program was proved in 82 steps with three user interactions.

4 An Alternative Formulation of Secure Information Flow in Dynamic Logic

There is another approach which captures the definition of secure information flow in an even more natural way than (2):

> "Running two instances of p with equal low-security values and arbitrary high-security values, the resulting low-security values are equal too."

This can be rendered in DL as follows:

$$\forall\, \texttt{h}.\, \forall\, \texttt{h}'.\, \forall\, \texttt{l}.\, (\texttt{l} \doteq \texttt{l}' \to \langle \texttt{p}[\texttt{l}, \texttt{h}];\ \texttt{p}[\texttt{l}', \texttt{h}']\rangle \texttt{l} \doteq \texttt{l}') \tag{7}$$

It is easy to see the drawbacks of this formulation:

– The number of security-relevant program variables is doubled, therefore, the state space might increase considerably.
– The approach can be used as it is only when the two instances of p do not interfere, that is, p[l, h] uses *only* the variables l and h. Otherwise, the remaining environment must be preserved.
– Leakage via termination behaviour cannot be expressed in an obvious way.

On the other hand, this approach potentially has important advantages:

– Instantiation of r is not required. Hence, all secure programs in Table 1 can be proved secure without any user interaction.

– In certain cases programs need to leak some confidential information, in order to serve their intended purpose [12]. Formulations (2,4) would classify these as insecure programs. This is too restrictive when the leakage was intended. We can extend (7) to express *intended leakage* via a suitable precondition. For example, if the least significant bit of h is leaked intentionally, then we add "$\mathtt{h}\, mod\, 2 \doteq \mathtt{h}'\, mod\, 2$" to the precondition.
– Executing the two instances of p in parallel (*lockstep*), instead of sequentially first $\mathtt{p}[\mathtt{l}, \mathtt{h}]$ and then $\mathtt{p}[\mathtt{l}', \mathtt{h}']$, may lead to efficient proofs: after each step, information that is irrelevant for the security analysis at hand can be deleted.
– Expressing insecurity can be easily done by taking the negation of (7). However, instantiation will be needed on the variables.

It is future work to investigate the possibilities and limitations of this approach, but it seems likely that both formulations should be used in combination for different types of problems and programs.

5 Discussion and Future Work

In this paper we suggested an interactive theorem prover for program verification as a framework for checking secure information flow properties. We showed the feasibility of the approach by applying it to a number of examples taken from the literature. Even without any tuning of the prover, the examples could be mechanically checked with few user interactions. Within a short amount of time, we managed to handle non-trivial properties such as method calls (with side effects), loops, object types. The method allows also to prove *insecurity*.

Most approaches to secure information flow are based on static analysis methods using domain-specific logics. These have the advantage of being usually decidable in polynomial time. On the other hand, they must necessarily abstract away from the target program. This becomes problematic when dealing with complex target languages such as Java Card. By taking a theorem proving approach and Java Card DL, which fully models the Java Card semantics, we can prove any property that is provable in first-order logic. Our experiments indicate that the penalty in terms of verification cost might be tolerable.

Joshi and Leino [7] consider how security can be expressed in various logical forms, leading to a characterisation of security using a Hoare triple. This characterisation is similar to the one used here—with the crucial difference that their formula *contains* a Hoare triple, but it is *not a statement in Hoare logic*, and thus cannot be plugged directly into a verification tool based on Hoare logic. Thus, the greater expressivity of dynamic logic has important advantages over Hoare logic in this context. We can provide mechanized, partially automated proofs for Java Card as target language.

In order to treat more realistic examples, we plan a number of improvements: on the side of the logic modeling, it might be useful to avoid existential quantification over the result values r in (2,4). In dynamic logic there are possibilities to do this, but they make proof obligations more complicated. It is not clear whether quantifier elimination over r will turn out to be a problem, because one can symbolically execute the programs *before* quantifier elimination. In addition, the KeY system will soon feature *metavariables*, by which instantiation can be delayed and handed over to an automated theorem prover for first order logic.

An open question is how this approach would scale-up for more complex programs with several high and low variables. To reduce the number of variables (and thus the number of quantifiers) the idea of abstract variables as "functions of the underlying program variables" proposed in [7] might be useful.

The KeY system is currently used "as-is". It can and should be tuned and adapted to security analysis, for example, by the addition of proof rules akin to the compositional rules offered by type systems.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.
3. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
4. B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proc. VERIFY Workshop at FLoC, Copenhagen, Denmark*, 2002. `i12www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz`.
5. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
7. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
8. J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460, pages 412–425, 2002.
9. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.
10. J. Rushby. Security requirements specifications: How and what? In *Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.
11. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communication*, 21(1), Jan. 2003.
12. A. C. M. Steve Zdancewic. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop, Cape Breton, Canada*, pages 15–23, 2001.
13. D. M. Volpano. Safety versus secrecy. In *Static Analysis Symposium*, pages 303–311, 1999.